第 8 章

Spring Boot 的监控之旅

监控是一个系统长期运营的必要保障，我们可以做一个这样的假设，当马路上不再有监控设备时，许多违章违纪的车辆将会钻法律的空子，不受法律的管理，长期这样，交通秩序将不再得到保障。而对于软件系统来说，监控同样必不可少，它可以在系统出现问题的时候自动提示系统维护人员，可以使出现的问题及时得到修复。本章笔者将带领大家学习 Spring Boot 常用的监控。

8.1 使用 actuator 监控

8.1.1 actuator 是什么

在 Spring Boot 的众多 Starter POMs 中有一个特殊的模块，不同于其他模块大多用于开发业务功能或连接一些其他外部资源，完全是一个用于暴露自身信息的模块，主要用于监控与管理，它就是 spring-boot-starter-actuator。

spring-boot-starter-actuator 模块的实现对于实施微服务的中小团队来说，可以有效地减少监控系统在采集应用指标时的开发量。当然，它并不是万能的，有时我们需要对其做一些简单的扩展来帮助我们实现自身系统个性化的监控需求。下面将详细介绍关于 spring-boot-starter-actuator 模块的内容，包括它原生提供的端点以及一些常用的扩展和配置方式。

8.1.2 如何使用 actuator

在 Spring Boot 中使用 actuator 很简单，只需要将项目加入 spring-boot-starter-actuator 依赖即可。这里为了方便观察，也加入了 spring-boot-starter-web 依赖，如代码清单 8-1 所示。

代码清单 8-1 Spring Boot-Actuator 项目依赖文件代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

8.1.3 actuator 监控介绍

没有特殊需求的话，其实到这里已经配置完成了，已经为应用程序开启了很多 actuator 端点。在 Spring Boot 应用中内置了很多端点，当然也支持添加自己需要的端点。例如，health 端点就提供了应用程序的健康信息。

大部分 actuator 端点都是可以独立进行的，通过对端点的启用和禁用可以控制是否创建端点用于查看信息。Spring Boot 应用通过 JMX 或者 HTTP 公开端点，大多数应用程序都是使用 HTTP 暴露端点的，端点使用前缀/actuator 加上端点 ID 来访问。例如，在默认情况下，health 端点映射到/actuator/health。

表 8-1 是 actuator 暴露的端点。

表 8-1 actuator 暴露的端点

HTTP 方法	ID	描述	默认情况下是否启用
GET	auditevents	显示应用程序的审核事件信息	是
GET	beans	显示应用程序中所有 Spring Bean 的完整列表	是
GET	conditions	显示在配置和自动配置类上评估的条件以及它们匹配或不匹配的原因	是
GET	configprops	显示配置列表，包括默认配置	是
GET	env	显示 Spring 的环境变量	是
GET	flyway	显示已应用的任何 Flyway 数据库迁移	是
GET	health	查看应用健康信息	是
GET	httptrace	显示 HTTP 跟踪信息（默认情况下显示最后 100 个）	是
GET	info	获取应用程序定制信息，这些信息提供 info 打头的属性	是
GET	loggers	显示和修改应用程序中记录器的配置	是
GET	liquibase	显示已应用的任何 Liquibase 数据库迁移	是
GET	metrics	显示当前应用程序的“指标”信息	是

（续表）

HTTP 方法	ID	描述	默认情况下是否启用
GET	mappings	显示所有@RequestMapping 路径的整理列表	是
GET	scheduledtasks	显示应用程序中的计划任务	是
GET	sessions	允许从 Spring Session 支持的会话存储中检索和删除用户会话。注：Spring Session 不支持 Web 响应式编程	是
POST	shutdown	允许应用程序正常关闭	是
GET	threaddump	执行线程转储	否
GET	heapdump	返回 GZip 压缩 hprof 堆转储文件	是
GET	jolokia	通过 HTTP 公开 JMX bean（当 Jolokia 在类路径上时，不适用于 WebFlux）	是
GET	logfile	返回日志文件的内容（如果已设置 logging.file 或 logging.path 属性）。支持使用 HTTP Range 标头来检索部分日志文件的内容	否
GET	prometheus	可以由 Prometheus 服务器抓取的格式公开指标	是

虽然大部分端点在默认情况下都是启用状态，但是在 Spring Boot 应用中，默认只开启 info 端点和 health 端点。其余端点都需要通过声明属性来开启，如代码清单 8-2 所示。

代码清单 8-2　开启全部端点相关代码

```
management.endpoints.web.exposure.include= *
```

通过以上设置可以开启所有默认启用的端点。当然，我们也可以根据 ID 指定开启端点，如代码清单 8-3 所示。

代码清单 8-3　开启局部端点相关代码

```
management.endpoints.web.exposure.include= heapdump,env
```

如果想要开启 shutdown 端点，那么可以使用如下配置使 shutdown 端点生效，如代码清单 8-4 所示。

代码清单 8-4　开启 shutdown 端点相关代码

```
management.endpoint.shutdown.enabled = true
```

其实，在 Spring Boot 应用程序中，启动项目后在日志上就可以看到开启的 Web 端点，如图 8-1 所示。

还有一种查看方式，就是通过 HTTP 请求访问/actuator，如图 8-2 所示。

图 8-1　Spring Boot-Actuator 项目通过启动日志查看 actuator 端点

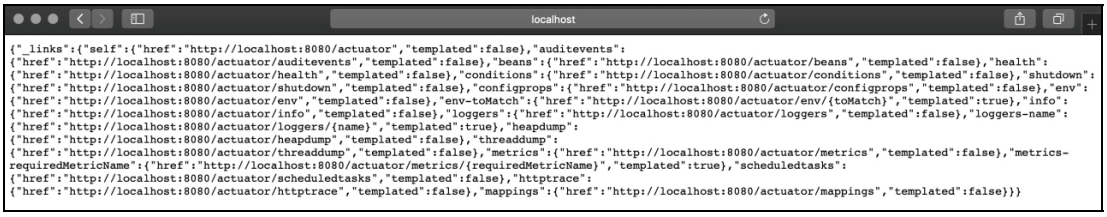

图 8-2　Spring Boot-Actuator 项目通过 HTTP 请求查看 actuator 端点

8.1.4　保护 HTTP 端点

Spring Boot 应用程序提供的 actuator 端点虽然为我们提供了一定的便利，但若没有安全限制，则会有一定的风险，比如 shutdown 端点随意暴露的话，应用的启停就会被“坏人”利用。

这时我们可以像使用任何其他敏感 URL 一样注意保护 HTTP 端点。若存在 Spring Security，则默认使用 Spring Security 的内容协商策略来保护端点。例如，如果你希望为 HTTP 端点配置自定义安全性，只允许具有特定角色的用户访问它们，Spring Boot 提供了一些 RequestMatcher 可以与 Spring Security 结合使用的便捷对象。

在项目中加入 spring-boot-starter-security 依赖，如代码清单 8-5 所示。

代码清单 8-5　Spring Boot-Actuator 项目新增依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

当在应用程序中加入 spring-boot-starter-security 依赖后，再次在浏览器中访问 actuator 端点时，页面如图 8-3 所示。

因为我们没有为 Spring Security 设置用户密码，所以暂时无法登录。接下来，在配置文件中为 Spring Security 设置一个安全用户，如代码清单 8-6 所示。

代码清单 8-6　Spring Boot-Actuator 项目配置 Spring Security 安全用户

```
spring.security.user.name=admin
spring.security.user.password=123456
```

图 8-3　Spring Boot-Actuator 项目通过 HTTP 请求登录验证

重启项目，再次访问 actuator 端点时，输入正确的用户名和密码即可正常查看 actuator 端点的信息，与图 8-2 中的内容一样。

actuator 也支持如 7-2 小节那样，通过配置权限来决定哪些方法可以被符合权限的用户访问，新增 ActuatorSecurity 配置类，继承 WebSecurityConfigurerAdapter 并且重写了 configure()方法，如代码清单 8-7 所示。

代码清单 8-7　Spring Boot-Actuator 项目 ActuatorSecurity 类代码

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint()).
authorizeRequests()
                .anyRequest().hasRole("ENDPOINT_ADMIN")
                .and()
                .httpBasic();
    }
}
```

在 ActuatorSecurity 配置类中设置任意 actuator 端点可以被安全用户登录，但是安全用户需要具备 ENDPOINT_ADMIN 权限，如果安全用户没有权限，访问就会报 403 错误（权限不足），如图 8-4 所示。

图 8-4　Spring Boot-Actuator 项目 403 错误页面

接下来，在配置文件中为安全用户赋予 ENDPOINT_ADMIN 权限，如代码清单 8-8 所示。

代码清单 8-8　Spring Boot-Actuator 项目将安全用户赋予权限

```
spring.security.user.roles=ENDPOINT_ADMIN
```

重启项目，再次访问可以正常查看 actuator 端点信息。

当然，我们也可以设置无须身份验证即可访问所有执行器端点。将 ActuatorSecurity 配置类的 @Configiration 注释上，其实就是取消 ActuatorSecurity 配置，新建 ActuatorNoSecurity 配置类。这里配置所有用户可以访问 actuator 端点，如代码清单 8-9 所示。

代码清单 8-9　Spring Boot-Actuator 项目无须安全验证配置类

```
@Configuration
public class ActuatorNoSecurity extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint()).
authorizeRequests()
                .anyRequest().permitAll();
    }
}
```

重启项目后，即使加入了 Spring Security，也无须身份验证就可以访问 actuator 端点。

8.1.5　健康信息

health 端点是查看 Spring Boot 应用程序健康状况的端点，如果没有特殊设置，显示的信息就比较少，如下所示：

```
{"status":"UP"}
```

我们可以通过在配置文件中设置 management.endpoint.health.show-details 来决定 health 端点的细节信息是否展示。以下为 health 端点的细节属性。

- never：细节信息详情永远都不展示。
- when-authorized：细节详情只对授权用户显示。
- always：细节详情显示给所有用户。

属性默认值为 never，当存在授权用户时，如果一个用户处于一个或者多个端点的角色，则将被视为已经获得授权。如果端点没有配置角色，则认为所有经过身份验证的用户都已获得授权。我们可以使用 management.endpoint.health.roles 属性配置角色。

这里以 always 为例，在配置文件中加入配置，如代码清单 8-10 所示。

代码清单 8-10　Spring Boot-Actuator 项目加入配置

```
management.endpoint.health.show-details=always
```

重启项目，重新访问 http://localhost:8080/actuator/health，这时健康端点信息如下：

```
{"status":"UP","details":{"diskSpace":{"status":"UP","details":{"total":2
50790436864,"free":101807063040,"threshold":10485760}}}}
```

当然，详情开放不只是针对 health 端点，其他端点同样适用。

在这个测试项目中，可能看不到 health 端点的作用，因为这个项目中没有配置其他相关的信息。其实健康信息的内容是从 HealthIndicators 中收集应用程序中定义的所有 bean 中的上下文信息，其中包含一些自动配置的 HealthIndicators，也可以编写自己的健康信息 bean。Spring Boot 默认会自动配置以下 HealthIndicators。

- CassandraHealthIndicator：检查 Cassandra 数据库是否已启动。
- DiskSpaceHealthIndicator：检查磁盘空间是否不足。
- DataSourceHealthIndicator：检查是否可以获得连接的 DataSource。
- ElasticsearchHealthIndicator：检查 Elasticsearch 集群是否已启动。
- InfluxDbHealthIndicator：检查 InfluxDB 服务器是否已启动。
- JmsHealthIndicator：检查 JMS 代理是否已启动。
- MailHealthIndicator：检查邮件服务器是否已启动。
- MongoHealthIndicator：检查 Mongo 数据库是否已启动。
- Neo4jHealthIndicator：检查 Neo4j 数据库是否已启动。
- RabbitHealthIndicator：检查 Rabbit 服务器是否已启动。
- RedisHealthIndicator：检查 Redis 服务器是否已启动。
- SolrHealthIndicator：检查 Solr 服务器是否已启动。

如果不想在项目中使用这些安全检查，就可以使用 management.health.defaults.enabled 属性来禁用它们，比如要禁用 Rabbit 安全检查，可以做如下设置，如代码清单 8-11 所示。

代码清单 8-11　Spring Boot-Actuator 项目禁用 Rabbit 服务器安全检查

```
management.health.rabbit.enabled=false
```

之前提到了自定义 HealthIndicators，接下来带领大家编写一个自定义的 HealthIndicators。其实要提供自定义健康状况信息，只需要编写一个实现 HealthIndicator 的类，重写 health 方法即可。这里编写一个简单的自定义 HealthIndicators，如代码清单 8-12 所示。

代码清单 8-12　Spring Boot-Actuator 项目自定义 HealthIndicators

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

    private final String defaultServerPort = "80";

    @Value("${server.port}")
    private String serverPort;

    @Override
    public Health health() {
```

```
            int errorCode = check();
            if (errorCode != 0) {
                return Health.down().withDetail("Error Code:", errorCode).build();
            }
            return Health.up().build();
        }

        public int check(){
            if(!defaultServerPort.equals(serverPort)){
                return 500;
            }
            return 0;
        }

    }
```

在 MyHealthIndicator 类中，其实只是做了一个检查，这里设置了一个默认的端口号。如果当前应用程序端口号不是默认端口号（80），就返回错误码 500；如果当前应用程序端口号为 8080，就启动项目，访问 http://localhost:8080/actuator/health，如下所示：

```
{"status":"DOWN","details":{"my":{"status":"DOWN","details":{"Error
Code:":500}},"diskSpace":{"status":"UP","details":{"total":250790436864,"free
":101805297664,"threshold":10485760}}}}
```

在输出信息中可以看到我们自定义的信息已经打印出来了。

8.1.6 自定义应用程序信息

在 actuator 端点中可以公开自定义信息，比如在配置文件中设置 info.*，如代码清单 8-13 所示。

代码清单 8-13 Spring Boot-Actuator 项目自定义 info 属性暴露

```
info.encoding = UTF-8
info.jdk.version = 1.8
```

重启项目，访问 http://localhost:8080/actuator/info 后即可看到，这里不再展示。

8.1.7 自定义管理端点路径

之前介绍了，actuator 端点在使用 HTTP 访问时需要使用前缀/actuator，其实这个前缀也可以根据需求自定义修改，只需要在配置文件中配置 management.endpoints.web.base-path 属性即可，如代码清单 8-14 所示。

代码清单 8-14 Spring Boot-Actuator 项目自定义管理端点路径

```
management.endpoints.web.base-path=/manage
```

修改后，再访问 health 端点，就由/actuator/health 变成了/manage/health。

actuator 是 Spring Boot 的重要特性，关于 actuator 的内容还有很多，提供端点能够做到的监控也有很多，具体可以查看官网对于 actuator 端点的介绍（Spring Boot 2.0.3 版本的官网地址：https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/actuator-api//html/）。

8.2　使用 Admin 监控

8.2.1　什么是 Spring Boot Admin

Spring Boot Admin 是 Spring Boot 项目的一个社区项目，主要用于管理和监控 Spring Boot 应用程序。通常来说，应用程序向我们的 Spring Boot Admin Server（通过 HTTP）直接注册信息或者 Spring Boot Admin Server 通过使用 Spring Cloud（例如 Eureka、Consul）服务发现收集 Client 信息。UI 只是 Spring Boot Actuator 端点上的一个 AngularJs 应用程序（2.x 版本后使用 Vue）。

8.2.2　设置 Spring Boot Admin Server

Spring Boot Admin 应用分为 Spring Boot Admin Server 应用和 Spring Boot Admin Client 应用。其中，Spring Boot Admin Server 应用用于收集 Spring Boot Admin Client 应用的信息并对其进行监控等。接下来，我们创建一个 Spring Boot Admin Server 应用。

创建 Spring Boot Admin Server 应用的过程大致分为两步，首先创建一个 Spring Boot 应用程序，在 pom 文件中加入依赖，如代码清单 8-15 所示。

代码清单 8-15　Spring Boot-Admin-Server 项目依赖文件代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

然后在 Spring Boot 应用程序启动类中加入@EnableAdminServer，引入 Spring Boot Admin Server 配置，如代码清单 8-16 所示。

代码清单 8-16　Spring Boot-Admin-Server 项目依赖文件代码

```
@Spring BootApplication
@EnableAdminServer
public class Chapter83Application {
```

```
    public static void main(String[] args) {
        SpringApplication.run(Chapter83Application.class, args);
    }
}
```

到这里，Spring Boot-Admin-Server 项目就已经配置完成了。启动项目可以看到如图 8-5 所示的页面。

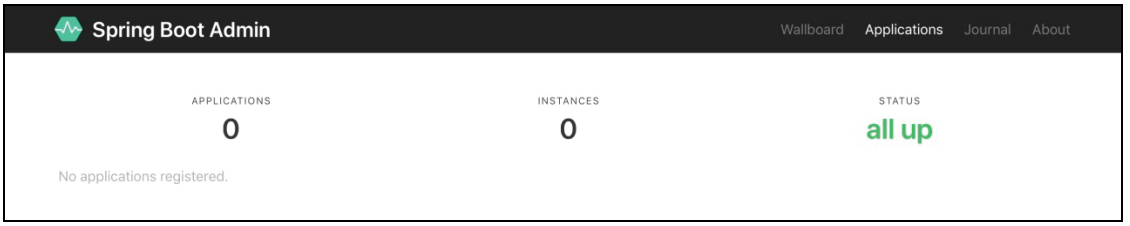

图 8-5　Spring Boot-Admin-Server 监控页面

由于还没有 Spring Boot Admin Client 应用注册到 Spring Boot-Admin-Server，因此现在实例数量还是 0。

8.2.3　Spring Cloud Eureka

可能到这里有人会问，这是一本讲解 Spring Boot 的书，为什么突然讲到 Spring Cloud 的组件 Eureka。因为 Spring Boot Admin Client 有一种方式是基于 Eureka 获取信息，所以这里介绍一下 Spring Cloud 的注册中心组件 Eureka。

引用官方的介绍：Eureka 是 Netflix 开发的服务发现框架，本身是一个基于 REST 的服务，主要用于定位运行在 AWS 域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。Spring Cloud 将它集成在子项目 spring-cloud-netflix 中，以实现 Spring Cloud 的服务发现功能。

通俗地理解，Eureka 就是一个注册中心。本小节仅对 Eureka Server 进行简单的介绍，感兴趣的读者可以去 Spring Cloud 官网（官网地址：https://cloud.spring.io/spring-cloud-static/Finchley.SR2/single/spring-cloud.html）查看更多关于 Eureka 的信息。

接下来，笔者带领大家创建一个 Eureka Server。新建项目，在 pom 文件中加入 spring-cloud-starter-netflix-eureka-server 依赖（使用 Spring Cloud Finchley.SR1 版本）。完整 pom 文件内容如代码清单 8-17 所示。

代码清单 8-17　Spring Cloud-Eureka-Server 项目依赖文件代码

```
    <?xml version="1.0" encoding="UTF-8"?>
    <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.springboot</groupId>
    <artifactId>chapter8-2</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>chapter8-2</name>
    <description>chapter8-2</description>

    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>Finchley.SR1</spring-cloud.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server
</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>

</project>
```

然后在启动类中加入注解@EnableEurekaServer，表明当前 Spring Boot 应用程序是一个 Eureka Server 程序，如代码清单 8-18 所示。

代码清单 8-18　Spring Cloud-Eureka-Server 项目启动类代码

```
@SpringBootApplication
@EnableEurekaServer
public class Chapter82Application {

    public static void main(String[] args) {
        SpringApplication.run(Chapter82Application.class, args);
    }
}
```

最后，我们只需要在配置文件中对 Eureka Server 应用程序进行配置，因为这里使用单节点 Eureka Server 应用，注意设置禁止向自己注册服务。配置文件如代码清单 8-19 所示。

代码清单 8-19　Spring Cloud-Eureka-Server 配置文件代码

```
server.port=8761
eureka.instance.hostname=localhost
eureka.client.service-url.defaultZone=http://${eureka.instance.hostname}:
${server.port}/eureka/

##禁止向自己注册
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

到这里，Eureka Server 应用配置完成了。启动项目，在浏览器中访问 http://localhost:8761，可以看到如图 8-6 所示的页面。

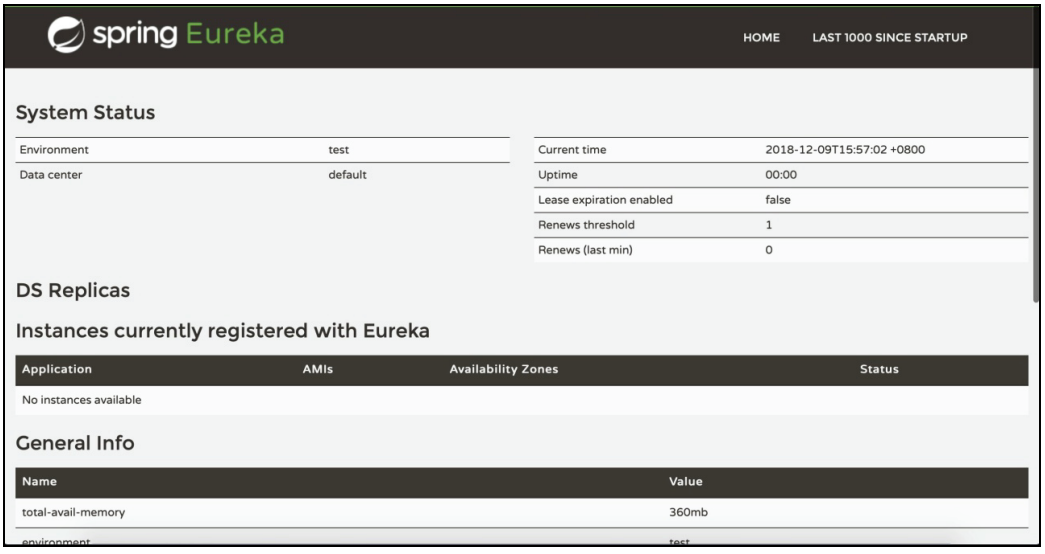

图 8-6　Spring Boot-Eureka Server 监控页面

与 Spring Boot Admin Server 一致，目前还没有实例注册进来，所以现在实例是空的，稍后会继续使用。

8.2.4　Spring Boot Admin Client 的使用

前面介绍了，使用 Spring Boot Admin Server 监控管理有两种方式，基于 Spring Cloud Discovery（Eureka 服务发现）或者对实例应用进行配置。接下来笔者带领大家分别对两种模式进行学习。

1. Spring Boot Admin 客户端

使用这种方式，所有需要使用 Spring Boot Admin Server 监控管理的应用程序都需要引入 spring-boot-admin-starter-client 依赖，如代码清单 8-20 所示。

代码清单 8-20　Spring Boot Admin Client 项目依赖文件代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.0.3</version>
</dependency>
```

接下来，在配置文件中配置 Spring Boot Admin Server 的地址，并且配置应用名称，以便在 Spring Boot Admin Server 页面查看。这里将端口号设置为 8081，如代码清单 8-21 所示。

代码清单 8-21　Spring Boot Admin Client 项目依赖文件代码

```
spring.boot.admin.client.url=http://localhost:8080
server.port=8081
spring.application.name=springboot-admin-client
```

启动 Spring Boot Admin Client 应用程序后，再来查看一下 Spring Boot Admin Server 监控页面，如图 8-7 所示。

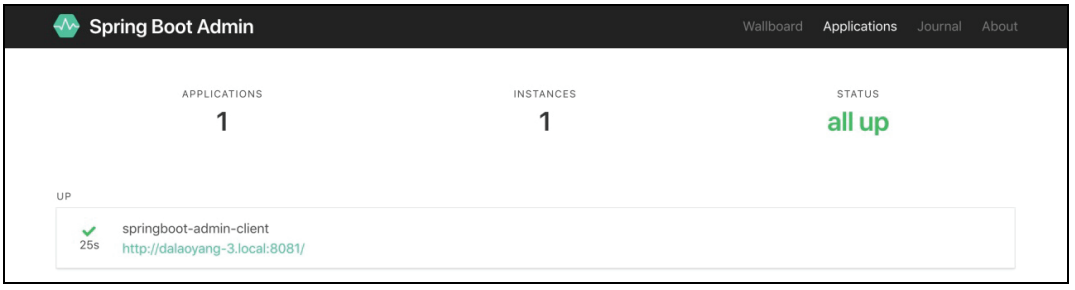

图 8-7　Spring Boot-Admin Server 监控页面

从图 8-7 中可以看到我们刚刚创建的应用实例已经注册到了 Spring Boot Admin Server 中，在 Spring Boot Admin Server 中会检测 Spring Boot Admin Client 实例的健康状况，其实就是检测 actuator 端点的 health 端点，因为当前应用实例状态为 up，所以状态显示为 all up。如果当前有任何应用不是 up 状态，就会显示状态为 down。

接下来单击实例，可以查看实例的详细信息，如图 8-8 所示。

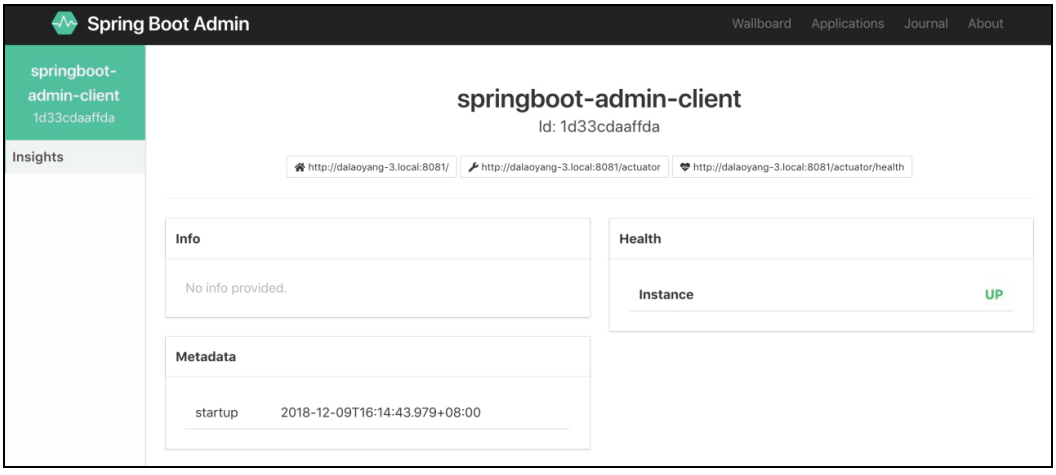

图 8-8　实例的详细信息

由于当前 Spring Boot Admin Client 应用程序没有为 actuator 开放更多的端点，因此这里仅能看到很少的信息。接下来我们为当前应用程序开放全部端点并且设置显示全部详细信息，然后查看 Spring Boot Admin Server 监控页面，如图 8-9 所示。

从图 8-9 中可以看到 Spring Boot-Admin Server 监控页面的数据随着开放的端点变多而变多，其实 Spring Boot-Admin Server 监控就是对 Spring Boot 应用程序的 actuator 端点进行一定的监控管理。

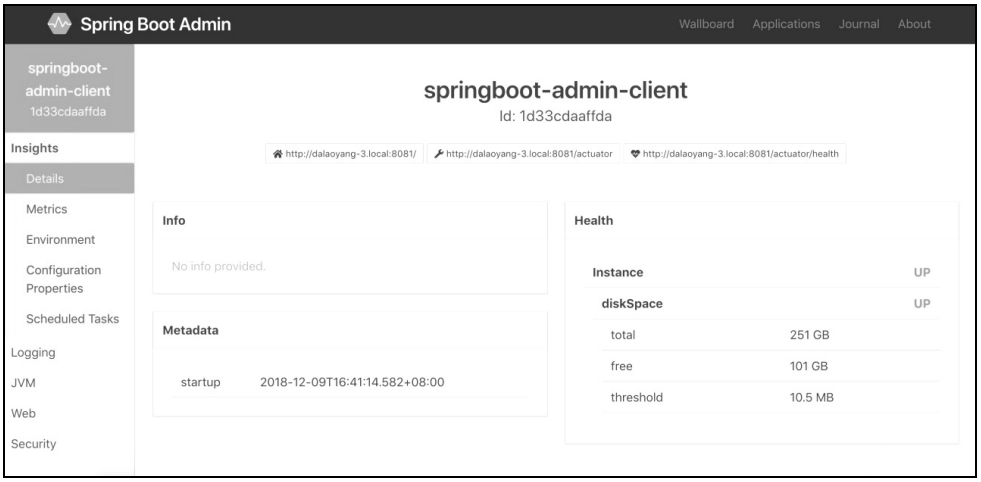

图 8-9　再次查看 Spring Boot-Admin Server 监控页面

2. 基于服务发现

接下来，我们使用 Eureka 的方式使用 Spring Boot-Admin Server 监控管理。新建项目，在项目中加入 spring-cloud-starter-netflix-eureka-client 依赖，如代码清单 8-22 所示。

代码清单 8-22　Spring Boot Eureka Client 项目依赖文件完整代码

```
   <?xml version="1.0" encoding="UTF-8"?>
   <project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <parent>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-parent</artifactId>
         <version>2.0.3.RELEASE</version>
         <relativePath/> <!-- lookup parent from repository -->
      </parent>
      <groupId>com.springboot</groupId>
      <artifactId>chapter8-5</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <packaging>jar</packaging>
      <name>chapter8-5</name>
      <description>chapter8-5</description>

      <properties>
         <java.version>1.8</java.version>
```

```
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client
</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Finchley.SR1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
```

```
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
</project>
```

在配置文件中配置 Eureka Server 的地址，并且开启全部 actuator 端点，设置端口号为 8082，如代码清单 8-23 所示。

代码清单 8-23　Spring Boot Admin Client 项目依赖文件代码

```
server.port=8082
spring.application.name=springboot-admin-client2
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
## eureka server 地址
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

然后将 Spring Boot Admin Server 应用程序加入 Eureka Client 依赖及配置，过程同上。全部配置完成后重启项目，先查看一下 Eureka Server 监控页面，如图 8-10 所示。

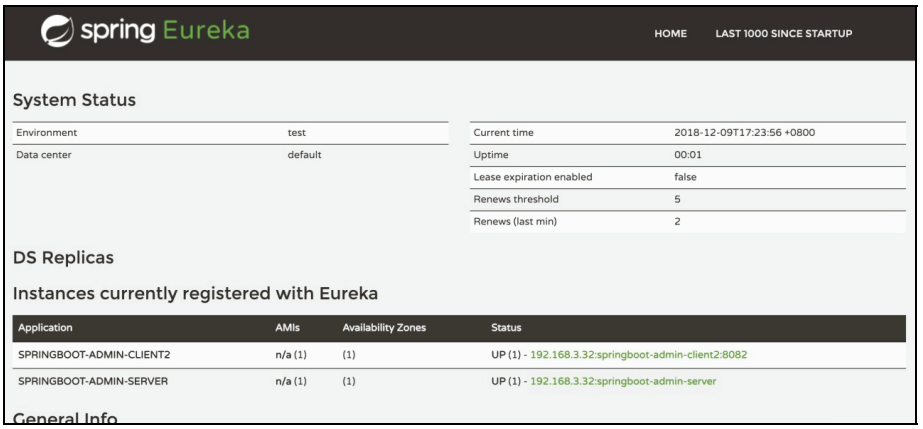

图 8-10　Spring Boot-Eureka Server 监控页面

可以看到实例部分已经存在两个使用 Eureka 的应用实例。接着查看 Spring Boot Admin Server 监控页面，如图 8-11 所示。

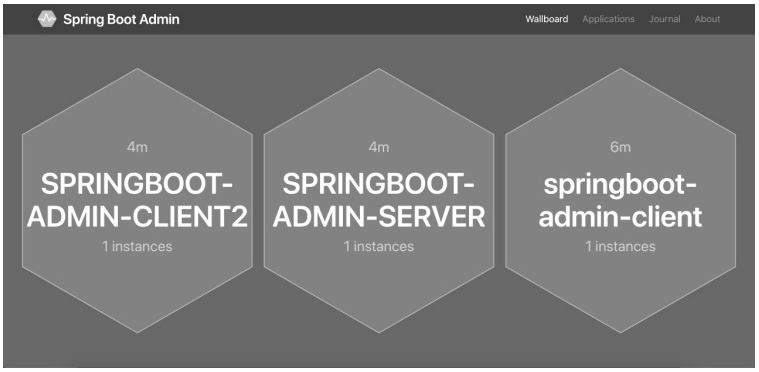

图 8-11　Spring Boot-Admin Server 监控页面

笔者分别介绍了使用 Spring Boot Admin 客户端和基于服务发现两种方式进行注册实例，都可以达到使用 Spring Boot Admin 监控的目的，这里笔者建议使用基于 Eureka 的方式，这样会更简单一些，无须基于每个应用频繁地配置。

8.2.5　安全验证

如果 Spring Boot Admin Server 监控页面可以随意查看，似乎不太安全。接下来笔者带领大家为 Spring Boot Admin Server 监控增加安全管理，在 pom 文件中加入 spring-boot-starter-security 依赖配置，如代码清单 8-24 所示。

代码清单 8-24　Spring Boot Admin Server 添加 Security 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

接下来加入一个配置类，使 actuator 端点可访问（这里设置登录用户可以查看全部端点），如代码清单 8-25 所示。

代码清单 8-25　Spring Boot Admin Server 添加 SecurityPermitAllConfig 配置代码

```
@Configuration
public class SecurityPermitAllConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().permitAll()
                .and().csrf().disable();
    }
}
```

在配置文件中配置用户名 admin、密码 123456，然后重启项目，访问 http://localhost:8080，Spring Boot Admin Server 为我们提供了友好的登录页面，如图 8-12 所示。

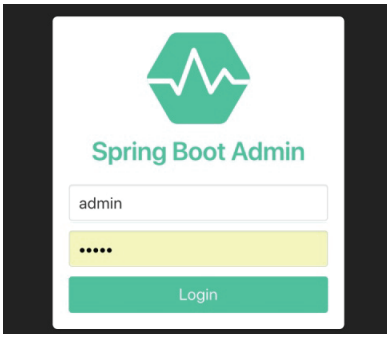

图 8-12　Spring Boot-Eureka Server 监控登录页面

使用用户名、密码登录后，就可以和之前一样查看 Spring Boot Admin Server 安全管理页面。

8.2.6　JMX-bean 管理

如果在 Spring Boot Admin Server 安全管理页面需要与 JMX-bean 交互，那么在应用程序中必须包含 Jolokia。由于 Jolokia 是基于 servlet 的，因此不支持响应式应用程序，WebFlux 在这里暂时不支持。使用 Jolokia 只需要引入 Jolokia 依赖，如代码清单 8-26。

代码清单 8-26　Jolokia 依赖代码

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
</dependency>
```

8.2.7　通知

Spring Boot Admin Server 不但提供了管理页面供我们查看，而且提供了一些通知，比如邮件通知、PagerDuty 通知、OpsGenie 通知等。本小节仅对邮件通知进行介绍。

邮件通知将作为使用 Thymeleaf 模板呈现的 HTML 电子邮件传递。如果需要启用邮件通知，就需要在项目中加入 spring-boot-starter-mail 依赖，如代码清单 8-27 所示。

代码清单 8-27　spring-boot-starter-mail 依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

在配置文件中新增邮箱相关配置信息，在后面的章节会介绍 Spring Boot 邮件发送的相关内容，这里不做过多解释。关于 Spirng Boot Admin 需要设置发送邮件地址和收件地址，如代码清单 8-28 所示。

代码清单 8-28　Spring Boot Admin Server 项目新增邮箱发送模板代码

```
spring.mail.host=smtp.qq.com
spring.mail.username=yangyang@dalaoyang.cn
spring.mail.password=邮件密码
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
spring.boot.admin.notify.mail.from=yangyang@dalaoyang.cn
spring.boot.admin.notify.mail.to=yangyang@dalaoyang.cn
```

重启项目后查看邮件，可以看到如图 8-13 所示的页面。

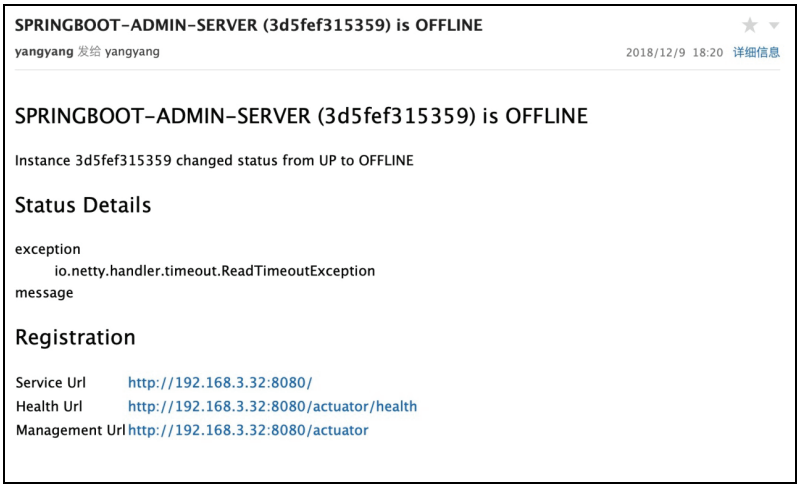

图 8-13　Spring Boot-Eureka Server 监控邮件发送

从图 8-13 中可以看到关于实例应用的状态变化、异常信息等，但是模板是英文的，如果需要自定义页面模板，那么可以新建一个 HTML，并且在配置文件中配置 spring.boot.admin.notify.mail.template。比如在 src/main/resources/文件夹下创建 status-changed.html 文件，然后在配置文件中配置 spring.boot.admin.notify.mail.template=classpath:/status-changed.html。以下是笔者根据官方提供的模板页面稍作修改的中文版，如代码清单 8-29 所示。

代码清单 8-29　Spring Boot Admin Server 项目新增邮箱相关配置代码

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <style>
        h1, h2, h3, h4, h5, h6 {
            font-weight: 400
        }
```

```
        ul {
            list-style: none
        }

        html {
            box-sizing: border-box
        }

        *, :after, :before {
            box-sizing: inherit
        }

        table {
            border-collapse: collapse;
            border-spacing: 0
        }

        td, th {
            text-align: left
        }

        body, button {
            font-family: BlinkMacSystemFont, -apple-system, "Segoe UI",
Roboto, Oxygen, Ubuntu, Cantarell, "Fira Sans", "Droid Sans", "Helvetica Neue",
Helvetica, Arial, sans-serif
        }

        code, pre {
            -moz-osx-font-smoothing: auto;
            -webkit-font-smoothing: auto;
            font-family: monospace
        }
    </style>
  </head>
  <body>
  <th:block th:remove="all">
    <!-- This block will not appear in the body and is used for the subject
-->
    <th:block th:remove="tag" th:fragment="subject">

        服务提醒: [[${instance.registration.name}]] (实例
ID:[[(${instance.id})]]) 状态变成了
        [[${event.statusInfo.status}]]
    </th:block>
```

```
   </th:block>
   <h1>服务名:<span th:text="${instance.registration.name}"/> (实例 ID:<span
th:text="${instance.id}"/>)
      状态变成了
      [[${event.statusInfo.status}]]
   </h1>
   <p>
      实例 <a th:if="${baseUrl}" th:href="@{${baseUrl + '/#/instances/' +
instance.id + '/'}}"><span
         th:text="${instance.id}"/></a>
      <span th:unless="${baseUrl}" th:text="${instance.id}"/>
      状态由 <span th:text="${lastStatus}"/> 变为 <span
th:text="${event.statusInfo.status}"/>
   </p>

   <h2>状态细节</h2>
   <dl th:fragment="statusDetails" th:with="details = ${details ?:
event.statusInfo.details}">
      <th:block th:each="detail : ${details}">
         <dt th:text="${detail.key}"/>
         <dd th:unless="${detail.value instanceof T(java.util.Map)}"
th:text="${detail.value}"/>
         <dd th:if="${detail.value instanceof T(java.util.Map)}">
            <dl th:replace="${#execInfo.templateName} :: statusDetails
(details = ${detail.value})"/>
         </dd>
      </th:block>
   </dl>

   <h2>注册信息</h2>
   <table>
      <tr th:if="${instance.registration.serviceUrl}">
         <td>服务地址</td>
         <td>
            <a th:href="${instance.registration.serviceUrl}"
th:text="${instance.registration.serviceUrl}"></a>
         </td>
      </tr>
      <tr>
         <td>健康地址</td>
         <td>
            <a th:href="${instance.registration.healthUrl}"
th:text="${instance.registration.healthUrl}"></a>
         </td>
```

```
        </tr>
        <tr th:if="${instance.registration.managementUrl}">
            <td>管理地址</td>
            <td>
                <a th:href="${instance.registration.managementUrl}"
th:text="${instance.registration.managementUrl}"></a>
            </td>
        </tr>
    </table>
    </body>
    </html>
```

使用自定义模板后，页面如图 8-14 所示。

图 8-14　Spring Boot-Eureka Server 监控邮件发送

笔者只是将对应英文修改成了中文，没有做过多修改，具体使用可以仔细考量需要得到的信息。

8.3　Prometheus+Grafana 监控

前面介绍了 Spring Boot Admin 监控 Spring Boot 应用程序，接下来将介绍由 Prometheus 结合 Grafana 搭建 Spring Boot 监控平台。

8.3.1　Prometheus 的安装

Prometheus（官网地址：https://prometheus.io/）是一个开源的系统监控和报警的工具包，最初由 SoundCloud 发布。Prometheus 通过在这些目标上抓取指标 HTTP 端点来收集受监控目标的指标。

由于 Prometheus 以同样的方式公开数据，因此它也可以掠夺和监控自身的健康状况，感兴趣的读者可以查阅官方文档。

以 Linux 系统为例进行介绍，到 Prometheus 官方下载页（官网下载页地址：https://prometheus.io/download/）下载安装包，如代码清单 8-30 所示。

代码清单 8-30　下载 Prometheus 代码

```
    wget https://github.com/prometheus/prometheus/releases/download/v2.6.1/
prometheus-2.6.1.linux-amd64.tar.gz
```

下载完成后，解压文件，如代码清单 8-31 所示。

代码清单 8-31　解压 Prometheus 代码

```
    tar xvfz prometheus-*.tar.gz
```

8.3.2　Grafana 的安装

Grafana（官网地址：https://grafana.com/）是一个深度分析的可视化工具，可以将采集的数据进行可视化的展示。如图 8-15 所示是 Grafana 官网首页，可以看出 Grafana 是做图形化分析的工具。

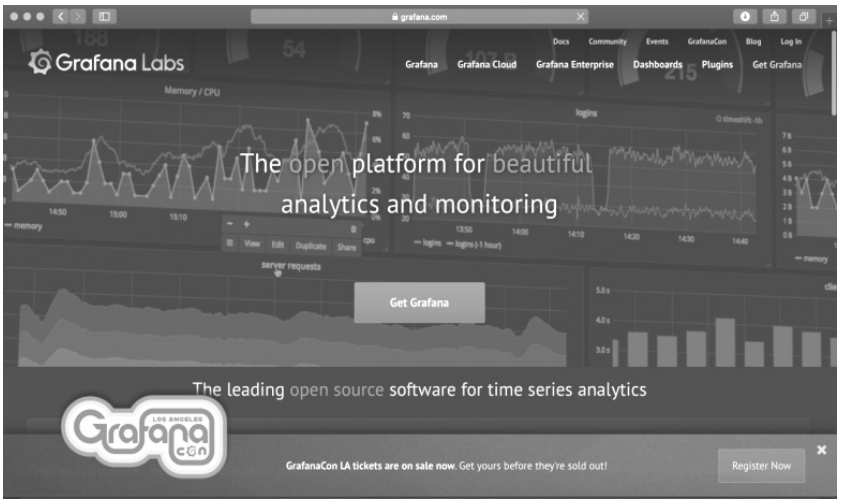

图 8-15　Grafana 官网首页

可以在 Grafana 官网下载页（官网下载页地址：https://grafana.com/grafana/download）下载 Grafana，里面详细介绍了各个操作系统如何安装，这里不再赘述。

8.3.3　Spring Boot 项目使用 Prometheus

新建项目，在项目中加入 micrometer-registry-prometheus 依赖，并且需要开启 actuator 端点，依赖文件如代码清单 8-32 所示。

代码清单 8-32　Prometheus 依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

在配置文件中加入配置应用名，并且开启全部 actuator 端点，如代码清单 8-33 所示。

代码清单 8-33　开启 actuator 端点代码

```
spring.application.name=chapter8-6
management.endpoints.web.exposure.include=*
```

在启动类中注入 MeterRegistryCustomizer 类，Chapter86Application 类完整内容如代码清单 8-34 所示。

代码清单 8-34　启动类代码

```
@SpringBootApplication
public class Chapter86Application {
    public static void main(String[] args) {
        SpringApplication.run(Chapter86Application.class, args);
    }

    @Value("${spring.application.name}")
    private String applicationName;

    @Bean
    MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
        return registry -> registry.config().commonTags("application",
applicationName);
    }
}
```

启动项目，访问 http://localhost:8080/actuator/prometheus，可以看到如图 8-16 所示的页面。

图 8-16　查看 prometheus 端点

8.3.4　Prometheus 配置

进入 Prometheus 安装目录，打开 prometheus.yml 文件，在 scrape_configs 中加入如下配置，如代码清单 8-35 所示。

代码清单 8-35　Prometheus 配置代码

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s
alerting:
  alertmanagers:
  - static_configs:
    - targets:
rule_files:
scrape_configs:
### 监控 prometheus
  - job_name: 'prometheus'
    static_configs:
    - targets: ['localhost:9090']
### 监控 SpringBoot 项目
  - job_name: 'chapter8-6'
    scrape_interval: 5s
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['127.0.0.1:8080']
```

启动 Prometheus，如代码清单 8-36 所示。

代码清单 8-36　启动 Prometheus 代码

```
./Prometheus
```

启动后，访问 http://localhost:9090/graph，如图 8-17 所示。

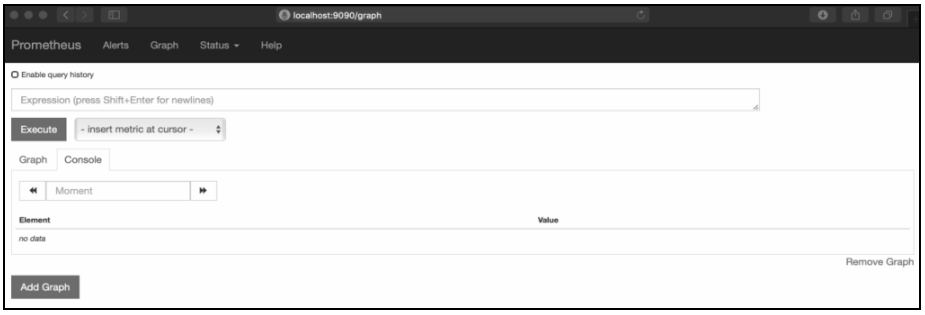

图 8-17　Prometheus 管理页面

单击导航栏 Status 中的 Targets，可以查看被 Prometheus 监控的应用，如图 8-18 所示。

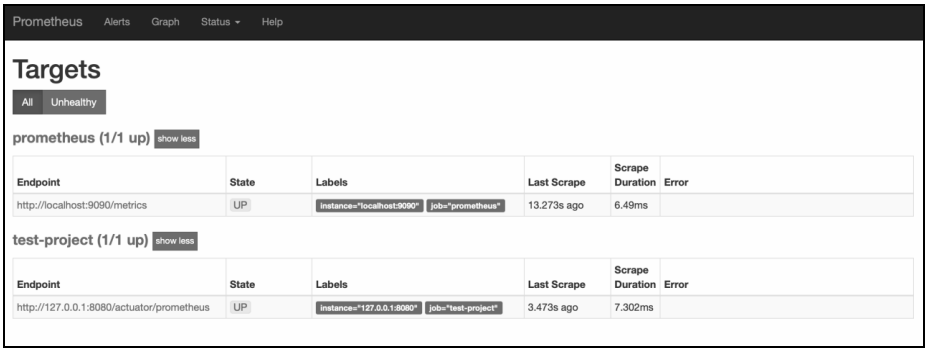

图 8-18　查看被 Prometheus 监控的页面

8.3.5　启动 Grafana

启动 Grafana，然后访问 http://localhost:3000/login，可以看到如图 8-19 所示的页面。

输入用户名 admin、密码 admin 进行登录。添加一个 Prometheus 数据源，在 URL 处配置 Prometheus 地址，因为都是本地安装，所以配置 http://localhost:9090 即可，配置完成后，单击 Save & Test 按钮，配置页如图 8-20 所示。

对于监控，这里推荐一个 Grafana 的看板，地址是 https://grafana.com/dashboards/6756，如图 8-21 所示。

图 8-19　Grafana 登录页面

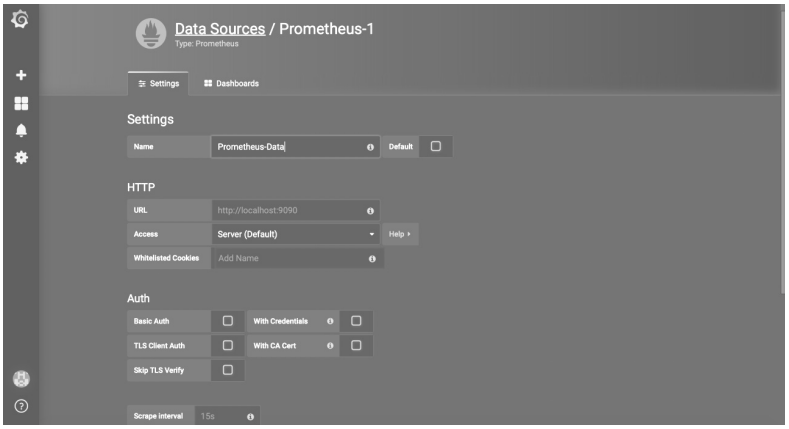

图 8-20　Grafana 配置数据源

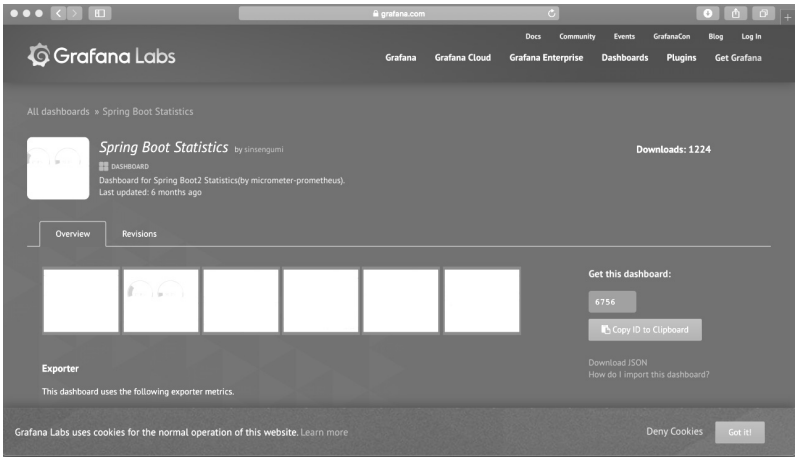

图 8-21　Grafana 查看 Dashboards 页面

单击 Grafana 导航栏的+号按钮，选择 import，这里输入 6756，然后进入如图 8-22 所示的页面。

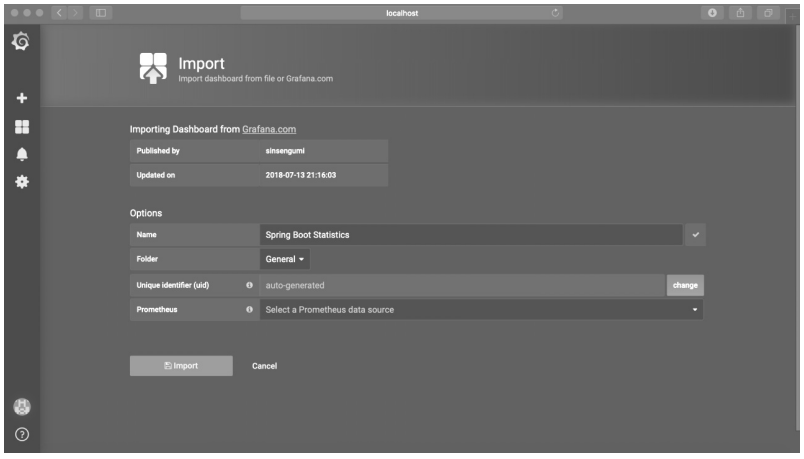

图 8-22　Grafana 导入 Dashboards 页面

Prometheus 选择刚刚添加的 Prometheus-Data，然后进入监控页面，如图 8-23 所示。

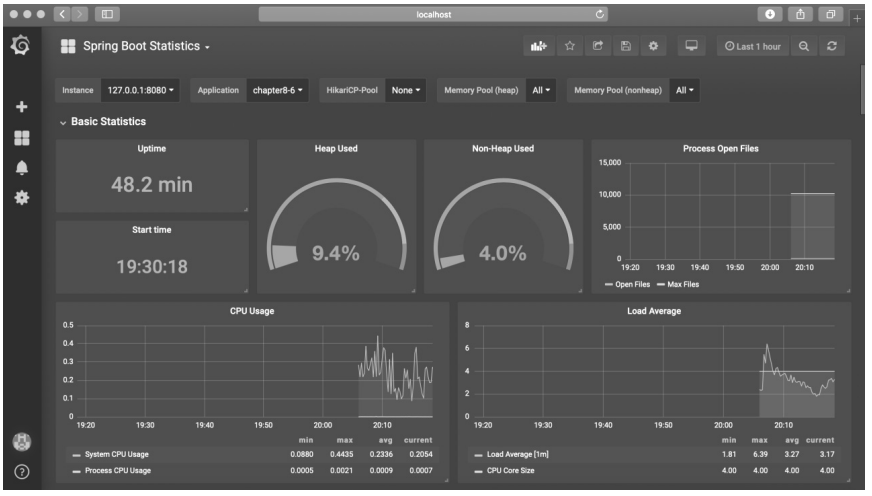

图 8-23　Grafana 查看指标页面

从图 8-23 中可以看到，已经监控到了我们想要监控的 Spring Boot 应用，这个监控页还有很多监控内容，比如 HTTP Statistics、Tomcat Statistics 等，如图 8-24 所示。

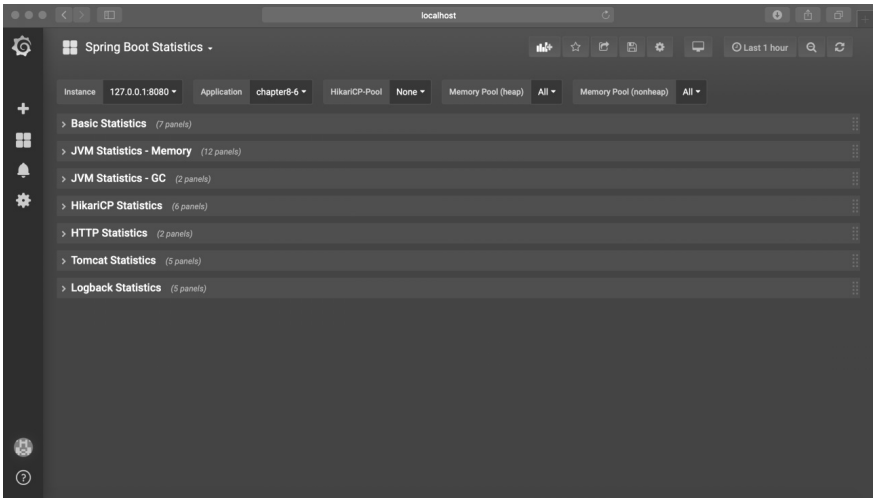

图 8-24　Grafana 监控内容

在这个监控中已经提供了足够多的监控查询，当然，在实际开发中可以添加自己需要监控的内容，根据项目场景添加即可。

8.4　小　　结

本章对 Spring Boot 服务的监控组件进行了一定的学习，从而保证服务的可用性。相信经过本章的学习，读者对监控已经有了一定了解，并且可以预防由项目意外瘫痪造成不必要的影响。

第 9 章

Spring Boot 的消息之旅

消息队列的英文名称为 Message Queue，通常简称为 MQ。MQ 是一种应用程序对应用程序的通信方法。消息队列是分布式系统中不可或缺的组件，主要解决应用解耦、异步消息、流量削峰等问题，实现高性能、高可用、可伸缩和最终一致性的架构。如今常用的开源消息队列组件有 RabbitMQ、Kafka、ActiveMQ、RocketMQ 等。

消息队列是典型的消费-生产者模式，生产者向消息队列生产消息，消费者可以从订阅的队列中读取消息。本章将对 Spring Boot 使用 RabbitMQ、Kafka、RocketMQ 消息队列进行介绍。

9.1 RabbitMQ 消息队列

RabbitMQ 是一个比较常用的消息队列，本小节将对它进行介绍，并介绍 Spring Boot 如何使用 RabbitMQ。

9.1.1 RabbitMQ 介绍

RabbitMQ 是一个由 Erlang 开发的 AMQP（Advanced Message Queuing Protocol）开源实现。很多人可能并不知道什么是 AMQP。AMQP 是一个提供统一服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息中间件设计。基于此协议的客户端与消息中间件可以传递消息，并不受客户端/中间件不同产品、不同开发语言等条件的限制。

RabbitMQ 是由 RabbitMQ Technologies Ltd 开发并且提供商业支持的。该公司在 2010 年 4 月被 SpringSource（VMWare 的一个部门）收购。在 2013 年 5 月被并入 Pivotal。其实 VMWare、Pivotal 和 EMC 本质上是一家的。不同的是，VMWare 是独立上市子公司，而 Pivotal 整合了 EMC 的某些资源，现在并没有上市。

RabbitMQ 支持多种客户端，如 Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP 等，支持 Ajax。用于分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面都有不错的表现。并且，正如 RabbitMQ 官网（官网地址：http://www.rabbitmq.com/）介绍的，RabbitMQ 在全球范围内在小型初创公司和大型企业中进行了超过 35 000 次 RabbitMQ 生产部署，是最受欢迎的开源消息代理。RabbitMQ 很轻量级，易于在内部和云中部署。它支持多种消息传递协议。RabbitMQ 可以部署在分布式和联合配置中，以满足高规模、高可用性的要求。

9.1.2　RabbitMQ 的几种角色

RabbitMQ 是一个消息代理，它的工作就是接收和转发消息。你可以把它想象成一个邮局：你把信件放入邮箱，邮递员就会把信件投递到收件人处。在这个比喻中，RabbitMQ 就扮演着邮箱、邮局以及邮递员的角色。

RabbitMQ 和邮局的主要区别在于它不处理纸张，而是接收、存储和发送消息（Message）这种二进制数据。

下面是 RabbitMQ 和消息所涉及的一些术语。

- 生产（Producing）的意思就是发送。发送消息的程序就是一个生产者（Producer）。我们一般用 P 来表示，如图 9-1 所示。

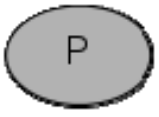

图 9-1　发送消息的生产者

- 队列（Queue）就是存在于 RabbitMQ 中邮箱的名称。虽然消息的传输经过了 RabbitMQ 和应用程序，但是它只能存储于队列中。实质上，队列就是一个巨大的消息缓冲区，它的大小只受主机内存和硬盘限制。多个生产者（Producers）可以把消息发送给同一个队列，同样，多个消费者（Consumers）也能够从同一个队列中获取数据。队列可以绘制，如图 9-2 所示。

图 9-2　队列

- 在这里，消费（Consuming）和接收（Receiving）是同一个意思。一个消费者（Consumer）就是一个等待获取消息的程序。我们把它绘制为 C，如图 9-3 所示。

图 9-3　消费者

需要指出的是，生产者、消费者、代理不需要待在同一个设备上。事实上，大多数应用确实不会将它们放在同一台机器上。

9.1.3 RabbitMQ 的几种模式

RabbitMQ 包含几种经典的消息传递模式，下面分别进行介绍。

1. 简单模式

在使用 RabbitMQ 消息队列的时候，最容易理解的就是点对点消息发送。我们可以这样理解这种模式，比如张三给李四发送信息，首先张三编辑短信，编辑完成后发送到信息中转站，然后由中转站转发送到李四的手机里。如图 9-4 所示，P 代表生产者，C 代表消费者，中间的盒子代表消费者保留的消息缓冲区，也就是队列，这种模式多用于聊天场景。

2. 工作队列模式

在发送消息时，还有这样一种场景，就是将一个消息发送给多个消费者。如果没有消息队列，我们就只能利用 HTTP 或者其他方式对多个消费者请求数据；如果使用消息队列，我们只需要将消息推送到工作队列中就可以解决问题。

工作队列（又称任务队列，Task Queues）是为了避免等待一些占用大量资源、时间的操作。当我们把任务（Task）当作消息发送到队列中时，一个运行在后台的工作者（Worker）进程就会取出任务，然后进行处理。当你运行多个工作者（Workers）时，任务就会在它们之间共享。

这个概念在网络应用中是非常有用的，多用于资源调度或抢红包等场景，它可以在短暂的 HTTP 请求中处理一些复杂的任务。如图 9-5 所示是工作队列模式的消息流转图。

图 9-4　简单模式的消息流转图

图 9-5　工作队列模式的消息流转图

3. 订阅模式

生产者（Producer）只需要把消息发送给一个交换机（Exchange）。交换机非常简单，它一边从生产者接收消息，一边把消息推送到队列。交换机必须知道如何处理它接收到的消息，是应该推送到指定的队列，还是推送到多个队列，或者直接忽略消息。这些规则是通过交换机类型（Exchange Type）来定义的。

有几个可供选择的交换机类型：直连（Direct）交换机、主题（Topic）交换机、头（Headers）交换机和扇型（Fanout）交换机。

可能到这里还是不好理解订阅模式与工作队列模式的区别，其实最简单的区别就是如果订阅模式有多个消费者，那么所有消费者都会收到消息，而工作队列模式只有一个消费者进行消费。订阅模式多用于广告、群聊等功能。如图 9-6 所示是订阅模式的消息流转图。

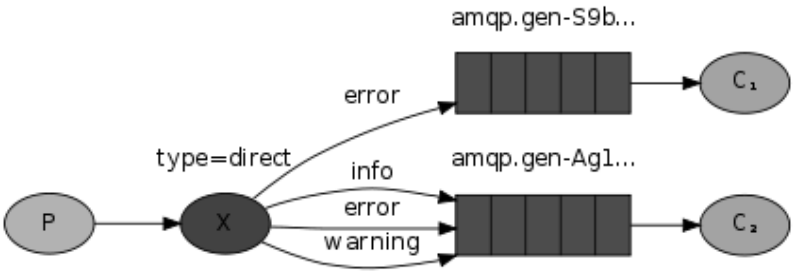

图 9-6　订阅模式的消息流转图

4. 路由模式

生产者将消息发送到交换机，在绑定队列和交换机的时候有一个路由 key，生产者发送的消息会指定路由 key，消息只会发送到 key 相同的队列，接着监听该队列消费者的消费信息。

路由模式很好理解，其实可以理解为订阅模式的特例，需要根据指定 key 来发布和订阅。一般来说，路由模式多用于项目中的报错信息。如图 9-7 所示是路由模式的消息流转图。

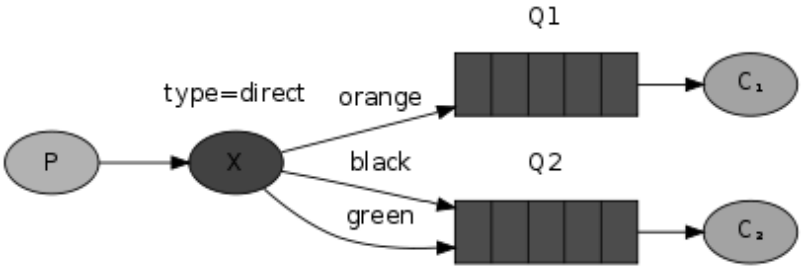

图 9-7　路由模式的消息流转图

5. topic 模式

topic 模式与路由模式大致相同，不同的是 topic 模式通过匹配符订阅多个主题的消息，比如：

- *(星号) 用来表示一个单词。
- #(井号) 用来表示任意数量（零个或多个）单词。

topic 模式的消息流转图如图 9-8 所示。

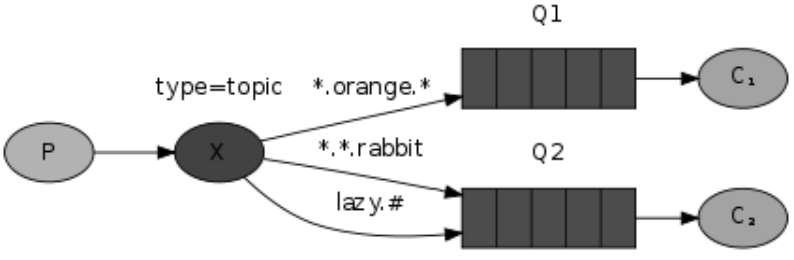

图 9-8　topic 模式的消息流转图

在这个例子里，我们发送的所有消息都是用来描述小动物的。发送的消息所携带的路由键是由 3 个单词组成的。路由键里的第一个单词描述的是动物的颜色，如图 9-8 所示的 orange（橙色）；

第二个单词是动物的种类，如图 9-8 所示的 rabbit（兔子）；第三个单词是动物的行为，如图 9-8 所示的 lazy（懒惰）。

为此，我们设置了 3 个绑定键，这 3 个绑定键可以总结为：

- Q1 队列对所有的橙色动物都感兴趣。
- Q2 队列则是对所有的兔子和所有懒惰的动物感兴趣。

举个例子，一个携带 quick.orange.rabbit 的消息将会被分别投递给 Q1 和 Q2 队列，携带 lazy.orange.elephant 的消息同样会投递给这两个队列。另一方面，携带 quick.orange.fox 的消息会投递给第一个队列，携带 lazy.brown.fox 的消息会投递给第二个队列。携带 lazy.pink.rabbit 的消息只会投递给第二个队列一次，即使它同时匹配第二个队列的两个绑定。携带 quick.brown.fox 的消息不会投递给任何一个队列。

如果我们违反约定，发送了一个携带一个单词或者 4 个单词（orange 或 quick.orange.male.rabbit）的消息，发送的消息不会投递给任何一个队列，并且会丢失掉。

但是，即使 lazy.orange.male.rabbit 有 4 个单词，还是会匹配最后一个绑定，并且投递到第二个队列。

6. 远程过程调用

RPC 是指远程过程调用。也就是说有两台服务器 A 和 B，一个应用部署在 A 服务器上，想要调用 B 服务器上应用提供的函数/方法，由于不在一个内存空间，因此不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。

一般在 RabbitMQ 中做 RPC 是很简单的。客户端发送请求消息，服务器回复响应的消息。为了接收响应的消息，我们需要在请求消息中发送一个回调队列。消息流转图如图 9-9 所示。

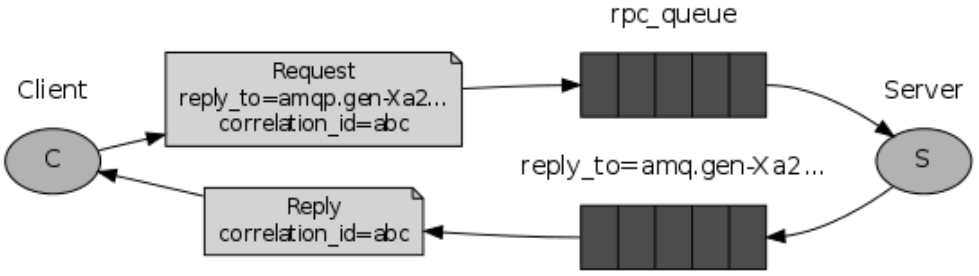

图 9-9　远程过程调用模式的消息流转图

9.1.4　Spring Boot 使用 RabbitMQ

在 Spring Boot 中使用 RabbitMQ 前需要启动 RabbitMQ。当启动 RabbitMQ 后，使用 RabbitMQ 大致分为如下几步：

（1）加入 RabbitMQ 依赖。
（2）配置 RabbitMQ 服务信息。
（3）编写消费者和生产者。

熟知上述步骤后，接下来我们新建项目，在项目中加入 spring-boot-starter-amqp 依赖，如代码清单 9-1 所示。

代码清单 9-1　RabbitMQ 项目引入 amqp 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

接下来，需要在配置文件中配置 RabbitMQ 服务信息，如代码清单 9-2 所示。

代码清单 9-2　RabbitMQ 项目配置 RabbitMQ

```
spring.rabbitmq.host=RabbitMQ 服务 IP
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=admin
```

一般来说，消息队列发送的数据都是实体对象，这里创建一个商品实体类 Goods 进行数据传输。Goods 实体内容如代码 9-3 清单所示。

代码清单 9-3　RabbitMQ 项目发送实体代码

```
public class Goods implements Serializable {
    private static final long serialVersionUID = 6629065135155452917L;
    private Long goodsId;
    private String goodsName;
    private String goodsIntroduce;
    private Double goodsPrice;

    ... //省略 set、Get 方法
}
```

这里以使用 RabbitMQ 的简单消息发送、Topic 转发模式消息发送和 Fanout Exchange 模式消息发送 3 种为例，使用 Spring Boot 操作 RabbitMQ 进行消息发送和接收。

1. 简单消息发送

简单消息发送很好理解，前面已经介绍过了，其实就是发送者将消息发送到消息队列，再由消息队列转发到消费者。首先创建一个简单发送消息配置 DirectConfig 类，在类上加入注解 @Configuration，表明这是一个配置类。在类中定义一个点对点消息发送的常量值用作消息队列名称，同时向 Spring 注入 Queue 类并创建消息队列，完整内容如代码清单 9-4 所示。

代码清单 9-4 RabbitMQ 项目简单消息发送配置

```
@Configuration
public class DirectConfig {
    public static final String DIRECT_QUEUE = "direct.queue";

    @Bean
    public Queue directQueue() {
        // 第一个参数是队列名字， 第二个参数是指是否持久化
        return new Queue("direct.queue", true);
    }
}
```

接下来创建一个消息发送者 DirectSender 类。一般使用消息发送都是通过操作 AmqpTemplate 类，所以首先注入这个类。然后创建一个发送消息的方法，调用 convertAndSend()方法进行消息发送。DirectSender 类完整内容如代码清单 9-5 所示。

代码清单 9-5 RabbitMQ 项目简单消息发送生产者

```
@Component
public class DirectSender {
    private static final Logger log =
LoggerFactory.getLogger(DirectSender.class);

    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendDirectQueue() {
        Goods goods = new Goods(System.currentTimeMillis(),"测试商品","这是一
个测试的商品",98.6);
        log.info("简单消息已经发送");
        // 第一个参数是指要发送到哪个队列，第二个参数是指要发送的内容
        this.amqpTemplate.convertAndSend(DirectConfig.DIRECT_QUEUE, goods);
    }
}
```

消息发送者已经创建好了。接下来创建一个消息接收者 DirectReceiver 类，使用@RabbitListener 设置监听队列的名称，方法的参数就是接收到的实体对象，完整内容如代码清单 9-6 所示。

代码清单 9-6 RabbitMQ 项目简单消息发送消费者

```
@Component
public class DirectReceiver {
    private static final Logger log = LoggerFactory.getLogger
(DirectReceiver.class);

    // queues 是指要监听的队列的名字
```

```
    @RabbitListener(queues = DirectConfig.DIRECT_QUEUE)
    public void receiverDirectQueue(Goods goods) {
        log.info("简单消息接受成功，参数是：" + goods.toString());
    }
}
```

最后创建一个 DirectController 类进行测试，这里仅调用 directSender 生产者发送消息的方法，完整内容如代码清单 9-7 所示。

代码清单 9-7　RabbitMQ 项目测试类

```
@RestController
public class DirectController {
    @Autowired
    private DirectSender directSender;

    @GetMapping("directTest")
    public void directTest() {
        directSender.sendDirectQueue();
    }
}
```

启动应用，在浏览器中访问 http://localhost:8080/directTest，可以看到控制台如图 9-10 所示。

```
2019-01-20 11:25:46.769  INFO 6554 --- [nio-8080-exec-1] com.springboot.sender.DirectSender       : 简单消息已经发送
2019-01-20 11:25:46.874  INFO 6554 --- [cTaskExecutor-1] com.springboot.receiver.DirectReceiver   : 简单消息接受成功，参数是：Goods{goodsId=1547954746769, goodsName='测
试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
```

图 9-10　简单消息发送控制台

我们再来查看一下 RabbitMQ 管理页面，如图 9-11 所示。

图 9-11　简单消息发送 RabbitMQ 管理页面

刚刚创建的队列在 RabbitMQ 管理页面也可以查看。至此，简单消息发送成功。

2. Topic 转发模式消息发送

Topic 转发模式是通过设置主题的方式来进行消息发送和接收的，这里需要使用到 Route-key，创建一个 TopicConfig 类配置主题和交换机，完整内容如代码清单 9-8 所示。

代码清单 9-8 RabbitMQ 项目 Topic 模式配置

```
@Configuration
public class TopicConfig {
    public static final String TOPIC_QUEUE1 = "topic.queue1";
    public static final String TOPIC_QUEUE2 = "topic.queue2";
    public static final String TOPIC_EXCHANGE = "topic.exchange";

    @Bean
    public Queue topicQueue1() {
        return new Queue(TOPIC_QUEUE1);
    }
    @Bean
    public Queue topicQueue2() {
        return new Queue(TOPIC_QUEUE2);
    }
    @Bean
    public TopicExchange topicExchange() {
        return new TopicExchange(TOPIC_EXCHANGE);
    }
    @Bean
    public Binding topicBinding1() {
        return BindingBuilder.bind(topicQueue1()).to(topicExchange()).
with("topic.messge");
    }
    @Bean
    public Binding topicBinding2() {
        return BindingBuilder.bind(topicQueue2()).to(topicExchange()).
with("topic.#");
    }
}
```

生产者发送消息还是使用 convertAndSend()方法，不过需要在参数内设置 Route-key，完整内容如代码清单 9-9 所示。

代码清单 9-9 RabbitMQ 项目 Topic 模式生产者

```
@Component
public class TopicSender {
    private static final Logger log =
LoggerFactory.getLogger(TopicSender.class);
    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendTopicQueue() {
```

```
        Goods goods1 = new Goods(System.currentTimeMillis(),"测试商品 1","这是
第一个测试的商品",98.6);
        Goods goods2 = new Goods(System.currentTimeMillis(),"测试商品 2","这是
第二个测试的商品",100.0);
        log.info("TopicSender 已发送消息");
        // 第一个参数：TopicExchange 名字
        // 第二个参数：Route-Key
        // 第三个参数：要发送的内容
        this.amqpTemplate.convertAndSend(TopicConfig.TOPIC_EXCHANGE,
"topic.messge", goods1 );
        this.amqpTemplate.convertAndSend(TopicConfig.TOPIC_EXCHANGE,
"topic.messge2", goods2);
    }
}
```

这里其实是创建两个消费者，分别订阅不同主题的内容。测试的目的很简单，消费者 2 订阅的主题包含生产者发送的两条信息，消费者 1 只能收到其中的一条信息。消费者完整内容如代码清单 9-10 所示。

代码清单 9-10　RabbitMQ 项目 topic 模式消费者

```
@Component
public class TopicReceiver {
    private static final Logger log = LoggerFactory.getLogger
(TopicReceiver.class);

    @RabbitListener(queues = TopicConfig.TOPIC_QUEUE1)
    public void receiveTopic1(Goods goods) {
        log.info("receiveTopic1 收到消息：" + goods.toString());
    }
    @RabbitListener(queues = TopicConfig.TOPIC_QUEUE2)
    public void receiveTopic2(Goods goods) {
        log.info("receiveTopic2 收到消息：" + goods.toString());
    }
}
```

创建 TopicController 类进行调用测试，如代码清单 9-11 所示。

代码清单 9-11　RabbitMQ 项目 Topic 模式测试类

```
@RestController
public class TopicController {
    @Autowired
    private TopicSender topicSender;

    @GetMapping("topicTest")
```

```
    public void topicTest() {
        topicSender.sendTopicQueue();
    }
}
```

在浏览器中访问 http://localhost:8080/topicTest，查看控制台，如图 9-12 所示。至此，Topic 转发模式消息发送已经完成。

图 9-12　Topic 模式控制台

3. Fanout Exchange 模式消息发送

Fanout Exchange 模式消息发送是指广播型消息发送，订阅了这个交换机的消息都会收到消息内容。Fanout Exchange 模式与 Topic 模式有些类似，但是不需要设置 Route-key。完整内容如代码清单 9-12 所示。

代码清单 9-12　RabbitMQ 项目 Fanout Exchange 模式配置

```
@Configuration
public class FanoutConfig {
    public static final String FANOUT_QUEUE1 = "fanout.queue1";
    public static final String FANOUT_QUEUE2 = "fanout.queue2";
    public static final String FANOUT_EXCHANGE = "fanout.exchange";
    @Bean
    public Queue fanoutQueue1() {
        return new Queue(FANOUT_QUEUE1);
    }
    @Bean
    public Queue fanoutQueue2() {
        return new Queue(FANOUT_QUEUE2);
    }
    @Bean
    public FanoutExchange fanoutExchange() {
        return new FanoutExchange(FANOUT_EXCHANGE);
    }
    @Bean
    public Binding fanoutBinding1() {
        return BindingBuilder.bind(fanoutQueue1()).to(fanoutExchange());
    }
    @Bean
    public Binding fanoutBinding2() {
```

```
            return BindingBuilder.bind(fanoutQueue2()).to(fanoutExchange());
        }
    }
```

消息发送者依然调用 convertAndSend()方法。这里需要注意，第二个参数设置为空。因为 Fanout 交换机不处理路由键，只是简单地将队列绑定到交换机上，每个发送到交换机的消息都会被转发到与该交换机绑定的所有队列上。生产者内容如代码清单 9-13 所示。

代码清单 9-13　RabbitMQ 项目 Fanout Exchange 模式生产者

```
    @Component
    public class FanoutSender {
        private static final Logger log = LoggerFactory.getLogger
(FanoutSender.class);
        @Autowired
        private AmqpTemplate amqpTemplate;

        public void sendFanoutQueue() {
            Goods goods = new Goods(System.currentTimeMillis(),"测试商品","这是一
个测试的商品",98.6);
            log.info("sendFanoutQueue 已发送消息");
            this.amqpTemplate.convertAndSend(FanoutConfig.FANOUT_EXCHANGE, "",
goods );
        }
    }
```

消费者只是创建两个消息队列来接收消息，因为使用的是同一个交换机，所以都会收到消息。消费者内容如代码清单 9-14 所示。

代码清单 9-14　RabbitMQ 项目 Fanout Exchange 模式消费者

```
    @Component
    public class FanoutReceiver {
        private static final Logger log =
LoggerFactory.getLogger(FanoutReceiver.class);

        @RabbitListener(queues = FanoutConfig.FANOUT_QUEUE1)
        public void receiveFanout1(Goods goods) {
            log.info("receiveFanoutQueue1 监听到消息：" + goods.toString());
        }
        @RabbitListener(queues = FanoutConfig.FANOUT_QUEUE2)
        public void receiveFanout2(Goods goods) {
            log.info("receiveFanoutQueue2 监听到消息：" + goods.toString());
        }
    }
```

创建 FanoutController 来调用消息发送，如代码清单 9-15 所示。

代码清单 9-15 RabbitMQ 项目 fanout 模式测试类

```
@RestController
public class FanoutController {
    @Autowired
    private FanoutSender fanoutSender;

    @GetMapping("fanoutTest")
    public void fanoutTest(){
        fanoutSender.sendFanoutQueue();
    }
}
```

在浏览器中访问 http://localhost:8080/fanoutTest，查看控制台，如图 9-13 所示。到这里，Fanout Exchange 模式消息发送已经完成。

```
2019-01-20 12:04:43.197  INFO 6723 --- [nio-8080-exec-4] com.springboot.sender.FanoutSender       : sendFanoutQueue已发送消息
2019-01-20 12:04:43.237  INFO 6723 --- [cTaskExecutor-1] com.springboot.receiver.FanoutReceiver   : receiveFanoutQueue2监听到消息: Goods{goodsId=1547957083197,
 goodsName='测试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
2019-01-20 12:04:43.237  INFO 6723 --- [cTaskExecutor-1] com.springboot.receiver.FanoutReceiver   : receiveFanoutQueue1监听到消息: Goods{goodsId=1547957083197,
 goodsName='测试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
```

图 9-13 Fanout Exchange 模式控制台

RabbitMQ 相关内容到这里就介绍完了。由于篇幅原因，不能面面俱到，只是列举了 3 个常用的方式来供读者参阅。

9.2 Kafka 消息队列

Kafka 是由 Apache 软件基金会开发的一个开源流处理平台，被誉为最高吞吐量的常用消息队列。本节我们来学习 Spring Boot 如何使用 Kafka 消息队列。

9.2.1 Kafka 介绍

Kafka（官网地址：http://kafka.apache.org/）是一种高吞吐量的分布式发布订阅消息系统，最初由 Linkedin 公司开发，于 2010 年底在 Github 首次开源，初始版本为 0.7.0。在 2011 年 7 月，Linkedin 公司将 Kafka 项目贡献给 Apache，成为 Apache 的孵化项目，在 2012 年 10 月，Kafka 从 Apache 孵化器正式毕业，成为 Apache 顶级项目。在 2014 年，为了 Kafka 更好地发展，几名 Kafka 的核心开发人员离开了 Linkedin，成交了 Confluent 公司，继续推进 Kafka 的发展。

在业界，Kafka 无疑是最高吞吐量的分布式流处理平台，并且它也是一个优秀的分布式 MQ，其通过 Zookeeper 实现了高可用。同时，Kafka 可以做的事情非常多，下面详细介绍。

1. 消息

Kafka 可以很好地替代传统的消息代理。消息代理的使用有多种场景，如将数据处理与数据生成器分离、缓冲未处理的消息等。与大多数消息传递系统相比，Kafka 具有更好的吞吐量、内置分区、复制和容错功能，这使其成为大规模消息处理应用程序的理想解决方案。通常消息传递使用较低的吞吐量，但可能要求较低的端到端延迟，Kafka 提供强大的持久性来满足这一要求。

在这个领域，Kafka 可与传统的消息传递系统（如 ActiveMQ 或 RabbitMQ）相媲美。

2. 网站活动跟踪

Kafka 的初始用例是能够将用户活动跟踪管道重建为一组实时发布 - 订阅源。这意味着网站活动（页面查看、搜索或用户可能采取的其他操作）将发布到中心主题，每个活动类型包含一个主题。这些订阅源提供了一系列用例，包括实时处理、实时监控以及加载到 Hadoop 或离线数据仓库系统以进行离线处理和报告。因为每个用户页面视图生成了许多活动消息，活动跟踪的数据量通常非常大。

3. 度量

Kafka 通常用于监控数据。这涉及从分布式应用程序聚合统计信息，并且从中生成可操作的集中数据源。

4. 日志聚合

许多人使用 Kafka 作为日志聚合解决方案的替代品。日志聚合通常从服务器收集物理日志文件，并将它们放在中央位置（可能是文件服务器或 HDFS）进行处理。Kafka 从这些日志中提取信息，并将日志或事件数据更清晰地抽象为消息流。这样可以更低延迟的处理并更容易支持多个数据源和分布式数据消耗。与 Scribe 或 Flume 等以日志为中心的系统相比，Kafka 提供了同样出色的性能，由于复制而具有更强的耐用性保证，以及更低的端到端延迟。

5. 流处理

许多 Kafka 用户在处理由多个阶段组成的管道时处理数据，其中原始输入数据从 Kafka 主题中消费，然后聚合、修饰或通过其他方式转换为新主题，以供进一步消费或后续处理。例如，用于推荐新闻文章的处理管道可以从 RSS 订阅源抓取文章内容并将其发布到“文章”主题；进一步处理可能会对此内容进行规范化或把重复数据删除，并将已清理的文章内容发布到新主题；最终处理阶段可能会尝试向用户推荐此内容。此类处理管道基于各个主题创建实时数据流的图形。从 0.10.0.0 版本开始，这是一个轻量级但功能强大的流处理库，名为 Kafka Streams，在 Apache Kafka 中可用于执行上述数据处理。除了 Kafka Streams 之外，其他开源流处理工具包括 Apache Storm 和 Apache Samza。

6. 活动采购

事件源是一种应用程序设计风格，按照时间来记录状态的更改。Kafka 可以存储非常多的日志数据，使其成为以这种风格构建的应用程序的出色后端。

7. 提交日志

Kafka 可以从外部为分布式系统提供日志提交功能。该日志有助于在节点之间复制数据，采用重新同步机制可以从失败的节点恢复数据。Kafka 中的日志压缩功能有助于支持此用法。在这种用法中，Kafka 类似于 Apache BookKeeper 项目。

9.2.2 Spring Boot 使用 Kafka

在 Spring Boot 中使用 Kafka 的过程与使用 RabbitMQ 大致一致，分为如下几步：

（1）加入 Kafka 依赖。
（2）配置 Kafka 服务信息。
（3）编写消费者和生产者。

在使用前需要安装 Kafka 服务。本节 Spring Boot 使用 Kafka 消息队列仅以发送实体对象为例，发送的实体还是 9.1 节使用的商品实体 Goods 类。新建项目，首先在 pom 文件中加入 Kafka 依赖，如代码清单 9-16 所示。

代码清单 9-16 Kafka 项目依赖代码

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

接下来，在配置文件中配置 Kafka 生产者和消费者的信息，如代码清单 9-17 所示。

代码清单 9-17 Kafka 项目配置文件

```
### producer 配置
spring.kafka.producer.bootstrap-servers=Kafka 服务地址:9092

### consumer 配置
spring.kafka.consumer.bootstrap-servers=Kafka 服务地址:9092
spring.kafka.consumer.group-id=goods
spring.kafka.consumer.enable-auto-commit=true
spring.kafka.consumer.auto-offset-reset=latest

spring.kafka.template.default-topic=goods
```

创建一个生产者 KafkaSender 类，在 Kafka 消息队列中使用消息发送的时候操作的都是 KafkaTemplate 类。首先注入这个类，然后调用 send 方法，定义主题为 goods，完整内容如代码清单 9-18 所示。

代码清单 9-18　Kafka 项目生产者代码

```
@Component
public class KafkaSender {
    private static final Logger log = LoggerFactory.getLogger
(KafkaSender.class);
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void send() {
        Goods goods = new Goods(System.currentTimeMillis(),"测试商品","这是一
个测试的商品",98.6);
        log.info("KafkaSender 已发送消息");
        kafkaTemplate.send("goods", goods.toString());
    }
}
```

创建一个消费者 KafkaReceiver 类，使用 Kafka 消息队列消费者的时候，其实是使用 @KafkaListener 注解来监听对应的主题。经过 9.1 节的学习，读者应该大致有了思路，这里只不过是使用的类或注解不同而已。完整 KafkaReceiver 类内容如代码清单 9-19 所示。

代码清单 9-19　Kafka 项目消费者

```
@Component
public class KafkaReceiver {
    private static final Logger log = LoggerFactory.getLogger
(KafkaReceiver.class);

    @KafkaListener(topics = "goods")
    public void send(ConsumerRecord<?, ?> record) {
        Optional<?> kafkaMessage = Optional.ofNullable(record.value());
        if (kafkaMessage.isPresent()) {
            Object messge = kafkaMessage.get();
            log.info("【KafkaListener 监听到消息】" + messge);
        }
    }
}
```

最后创建一个 KafkaController 类进行调用消息发送，如代码清单 9-20 所示。

代码清单 9-20　Kafka 项目测试类代码

```
@RestController
public class KafkaController {
    @Autowired
    private KafkaSender kafkaSender;
```

```
    @GetMapping("testKafka")
    public void testKafka(){
        kafkaSender.send();
    }
}
```

启动项目，在浏览器中访问 http://localhost:8080/testKafka，然后查看控制台，如图 9-14 所示。

```
2019-01-20 12:50:21.989  INFO 6877 --- [nio-8080-exec-1] o.a.kafka.common.utils.AppInfoParser     : Kafka version : 1.0.1
2019-01-20 12:50:21.989  INFO 6877 --- [nio-8080-exec-1] o.a.kafka.common.utils.AppInfoParser     : Kafka commitId : c0518aa65f25317e
2019-01-20 12:50:22.590  INFO 6877 --- [ntainer#0-0-C-1] com.springboot.receiver.KafkaReceiver    : 【KafkaListener监听到消息】Goods{goodsId=1547959821955,
 goodsName='测试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
```

图 9-14　Kafka 项目控制台输出

到这里，使用 Kafka 消息队列已经完成了。其实 Kafka 可以做的不只是消息，还有很多方面的使用，具体内容可以查阅官网来进行学习。

9.3　RocketMQ 消息队列

9.3.1　RocketMQ 介绍

Apache RocketMQ（官网地址：http://rocketmq.apache.org）是由阿里巴巴集团开源的大型消息队列，现在已经贡献给了 Apache 开源基金会，同时是一个分布式消息传递和流媒体平台，具有低延迟、高性能、可靠性、万亿级容量和灵活的可扩展性。

接下来介绍 RocketMQ（Github 官网地址：https://github.com/apache/rocketmq）的 Github，它提供了多种功能：

- 发布/订阅消息模型。
- 定时的消息传递。
- 按时间或偏移量对消息进行追溯。
- 记录流媒体的中心。
- 大数据集成。
- 可靠的 FIFO 和严格的有序消息传递在同一队列中。
- 高效的推拉消费模式。
- 单个队列中的百万级消息累积容量。
- 多种消息传递协议，如 JMS 和 OpenMessaging。
- 灵活的分布式横向扩展部署架构。
- Lightning-fast 批处理消息交换系统。
- 各种消息过滤器机制，如 SQL 和 Tag。
- Docker 图像用于隔离测试和云隔离集群。
- 功能丰富的管理仪表板，用于配置、指标和监控。

RocketMQ 由 4 部分组成，分别是 name servers、brokers、producers 和 consumers，如图 9-15 所示。

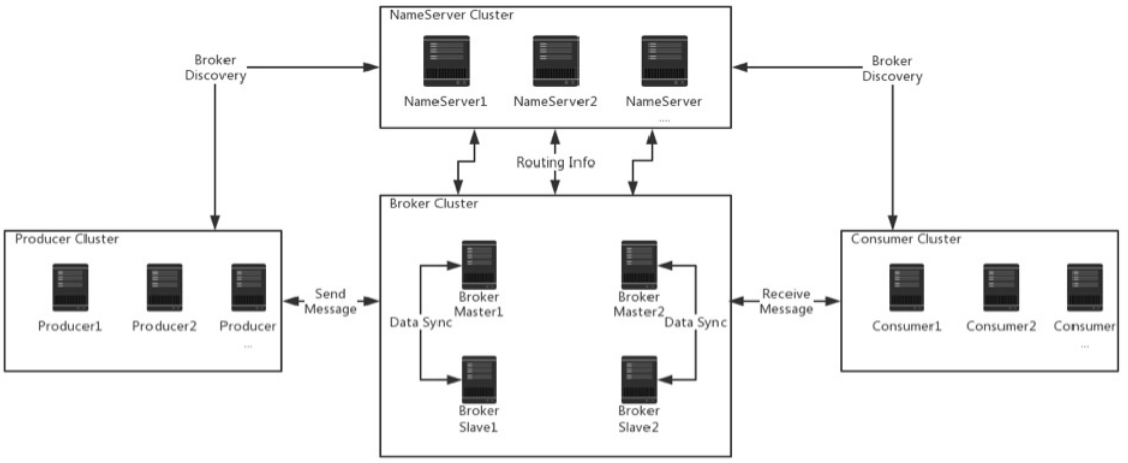

图 9-15　RocketMQ 组成图

9.3.2　Spring Boot 使用 RocketMQ

由于官网已经详细介绍了如何使用 RocketMQ，因此本小节仅以发送简单消息为例介绍 Spring Boot 如何使用 RocketMQ。

Spring Boot 使用 RocketMQ 消息队列大致分为三步：

（1）加入 RocketMQ 依赖。
（2）配置 RocketMQ 服务信息。
（3）编写生产者和消费者。

1. 加入 RocketMQ 依赖

新建项目，在 pom 文件中加入 RocketMQ 依赖，如代码清单 9-21 所示。

代码清单 9-21　RocketMQ 项目依赖代码

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.2.0</version>
</dependency>
```

2. 配置 RocketMQ 服务信息

在配置文件中加入 RocketMQ 服务的配置信息，如代码清单 9-22 所示。

代码清单 9-22 RocketMQ 项目配置文件

```
# 消费者的组名
apache.rocketmq.consumer.PushConsumer=PushConsumer

# 生产者的组名
apache.rocketmq.producer.producerGroup=Producer

# NameServer 地址
apache.rocketmq.namesrvAddr=localhost:9876
```

3. 编写生产者和消费者

这里以发送 10 条简单消息为例，创建一个生产者，这里使用的是默认生产者 DefaultMQProducer，在构建生产者的时候使用构造方法设置生产者的组名。使用 setNamesrvAddr() 方法设置 NameServer，如果有多个 NameServer，就使用逗号分隔。这里需要注意一点，生产者对象只调用一次 start 方法即可，不需要每次都调用。在构建消息体时设置 topic 和 tags。完整内容如代码清单 9-23 所示。

代码清单 9-23 RocketMQ 项目生产者代码

```
@Component
public class RocketMQSender {
    @Value("${apache.rocketmq.producer.producerGroup}")
    private String producerGroup;
    @Value("${apache.rocketmq.namesrvAddr}")
    private String namesrvAddr;
    private static final Logger log = LoggerFactory.getLogger
(RocketMQSender.class);

    public void defaultMQProducer() {
        DefaultMQProducer producer = new DefaultMQProducer(producerGroup);
        producer.setVipChannelEnabled(false);
        //指定 NameServer 地址，多个地址以 ; 隔开
        producer.setNamesrvAddr(namesrvAddr);
        try {
            producer.start();
            Message message = new Message("TopicTest", "push", "【发送消息】
".getBytes());
            StopWatch stop = new StopWatch();
            stop.start();

            for (int i = 0; i < 10; i++) {
```

```
                SendResult result = producer.send(message, new
MessageQueueSelector() {
                    @Override
                    public MessageQueue select(List<MessageQueue> mqs, Message
msg, Object arg) {
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                },1);
                log.info("发送响应：MsgId:" + result.getMsgId() + "，发送状态:" +
result.getSendStatus());
            }
            stop.stop();
            log.info("-----------------发送十条消息耗时：" +
stop.getTotalTimeMillis());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            producer.shutdown();
        }
    }

}
```

接下来编写一个消费者，其中的设置与生产者类似，这里就不做过多介绍了。完整消费者如代码清单 9-24 所示。

代码清单 9-24　RocketMQ 项目消费者代码

```
@Component
public class RocketMQReceiver {
    @Value("${apache.rocketmq.consumer.PushConsumer}")
    private String consumerGroup;
    @Value("${apache.rocketmq.namesrvAddr}")
    private String namesrvAddr;
    private static final Logger log = LoggerFactory.getLogger
(RocketMQReceiver.class);

    //@PostContruct 是 spring 框架的注解，在方法上加该注解会在项目启动的时候执行该方法，
也可以理解为在 spring 容器初始化的时候执行该方法
    @PostConstruct
    public void defaultMQPushConsumer() {
        //消费者的组名
```

```
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer
(consumerGroup);

        //指定 NameServer 地址，多个地址以 ; 隔开
        consumer.setNamesrvAddr(namesrvAddr);
        try {
            //订阅 PushTopic 下 Tag 为 push 的消息
            consumer.subscribe("TopicTest", "push");

            //设置 Consumer 第一次启动是从队列头部开始消费还是从队列尾部开始消费
            //如果非第一次启动，那么按照上次消费的位置继续消费
            consumer.setConsumeFromWhere(ConsumeFromWhere.
CONSUME_FROM_FIRST_OFFSET);
            consumer.registerMessageListener((MessageListenerConcurrently)
(list, context) -> {
                try {
                    for (MessageExt messageExt : list) {
                        //输出消息内容
                        log.info("messageExt: " + messageExt);
                        String messageBody = new String(messageExt.getBody());
                        //输出消息内容
                        log.info("【消费响应】: msgId : " + messageExt.getMsgId()
+ ",  msgBody : " + messageBody);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                    //稍后再试
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }
                //消费成功
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            });
            consumer.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //@PostContruct 是 spring 框架的注解,在方法上加该注解会在项目启动的时候执行该方法,
也可以理解为在 spring 容器初始化的时候执行该方法
    @PostConstruct
    public void defaultMQPushConsumer2() {
        //消费者的组名
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("aaa");

        //指定 NameServer 地址，多个地址以 ; 隔开
```

```
            consumer.setNamesrvAddr(namesrvAddr);
            try {
                //订阅 PushTopic 下 Tag 为 push 的消息
                consumer.subscribe("TopicTest", "push");

                //设置 Consumer 第一次启动是从队列头部开始消费还是从队列尾部开始消费
                //如果非第一次启动，那么按照上次消费的位置继续消费
                consumer.setConsumeFromWhere(ConsumeFromWhere.
CONSUME_FROM_FIRST_OFFSET);
                consumer.registerMessageListener((MessageListenerConcurrently)
(list, context) -> {
                    try {
                        for (MessageExt messageExt : list) {
                            //输出消息内容
                            log.info("---- messageExt: " + messageExt);
                            String messageBody = new String(messageExt.getBody());
                            //输出消息内容
                            log.info("----【消费响应】: msgId : " +
messageExt.getMsgId() + ",  msgBody : " + messageBody);
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                        //稍后再试
                        return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                    }
                    //消费成功
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                });
                consumer.start();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

最后编写一个 RocketMQController 类来调用生产者发送消息，如代码清单 9-25 所示。

代码清单 9-25　RocketMQ 项目测试类代码

```
    @RestController
    public class RocketMQController {
        @Autowired
        private RocketMQSender rocketMQSender;

        @GetMapping("testRocketmq")
        public void testRocketmq(){
```

```
            rocketMQSender.defaultMQProducer();
        }
    }
```

启动项目后，在浏览器中访问 http://localhost:8080/testRocketmq，如图 9-16 所示。

图 9-16　RocketMQ 项目控制台输出

RocketMQ 消息发送已经成功了。还有很多特性，具体可以查看官网教程学习。

9.4　消息队列对比

前面学习了常用的几个消息队列，本节从以下几方面对常用消息队列进行比较。

（1）关注度

首先我们从百度搜索指数来看本章介绍的 3 个消息队列的关注情况，如图 9-17 所示。

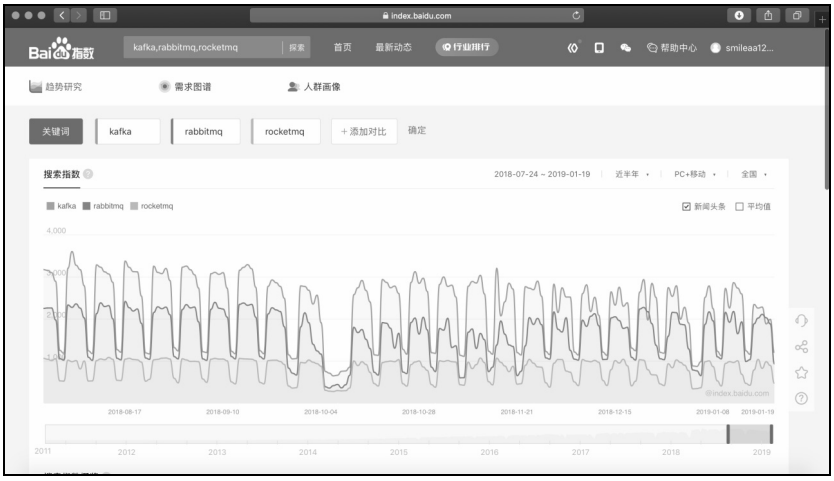

图 9-17　百度搜索指数队列搜索对比

从图 9-17 中可以看到，Kafka 消息队列远远领先于 RabbitMQ 消息队列，RocketMQ 最末。毕竟 RocketMQ 捐献给 Apache 基金会没有多久，这些老牌消息队列的关注度还是很高的。

接下来，我们来看谷歌趋势中的搜索指数对比，如图 9-18 所示。

图 9-18　谷歌趋势消息队列搜索对比

与百度搜索相近，在关注度方面，Kafka 最高，RabbitMQ 其次，RocketMQ 最末。

（2）成熟度

从成熟度来看，其实对 RocketMQ 并不是十分公平。毕竟和老牌消息队列相比，它还是一个初出茅庐的新手。所以就成熟度来说，Kafka 和 RabbitMQ 已经是成熟消息队列，而 RocketMQ 属于比较成熟的消息队列，毕竟自从开源给 Apache 基金会，已经发布几个版本了。

（3）社区活跃度

社区活跃度方面，虽然国人对 RocketMQ 的呼声很高，并且已经有大部分企业准备使用 RocketMQ，但是还是无法与老牌的消息队列社区比较。

（4）吞吐量

吞吐量方面，查看阿里中间件博客对三者的对比，结果如图 9-19 所示。

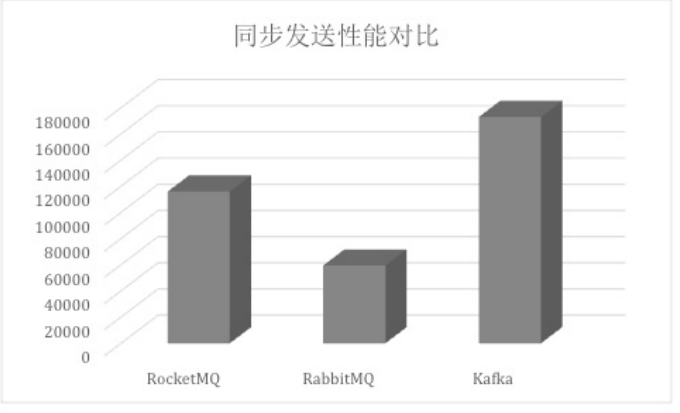

图 9-19　消息队列对比图

在同步发送场景中，3 个消息中间件的表现区别明显：

- Kafka 的吞吐量高达 17.3w/s，不愧是高吞吐量消息中间件的行业老大。这主要取决于它的队

列模式保证了写磁盘的过程是线性 IO，此时 broker 磁盘 IO 已达瓶颈。

- RocketMQ 也表现不俗，吞吐量为 11.6w/s，磁盘 IO %util 已接近 100%。RocketMQ 的消息写入内存后即返回 ack，由单独的线程专门做刷盘的操作，所有的消息均是顺序写文件的。
- RabbitMQ 的吞吐量为 5.95w/s，CPU 资源消耗较高。它支持 AMQP 协议，实现非常重量级，为了保证消息的可靠性，在吞吐量方面做了取舍。我们还做了 RabbitMQ 在消息持久化场景下的性能测试，吞吐量在 2.6w/s 左右。

测试结论：在服务端处理同步发送的性能上，Kafka>RocketMQ>RabbitMQ。

（5）可靠性

可靠性方面，RabbitMQ 最好；其次是 RocketMQ；Kafka 最差，会有丢失消息的情况。

（6）事务

在事务方面，只有 RocketMQ 提供了事务支持，其他消息服务都不提供事务支持。

（7）个人建议

如果现有消息队列用得很好，没有特别的性能要求，不需要重复造轮子。另外，要结合现有场景来使用，不要盲目追求吞吐量等指标。

9.5 小　　结

本章对 Spring Boot 使用消息队列进行了介绍，包括传统流行的 RabbitMQ 消息队列、Kafka 消息队列以及阿里巴巴经过多次“双 11”测试的 RocketMQ 消息队列，最后对消息队列进行了对比。相信经过学习，读者会对使用消息队列有所了解，从而在今后的工作中使用消息队列时不再迷茫，找到正确的方向。